Estruturas de Dados - ESP412

Profª Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Campus de Feira de Santana

carolsoko@ifba.edu.br

November 28, 2024

Algoritmos de Ordenação

Algoritmos de Ordenação

QuickSort

Ordenação por Pivoteamento (*QuickSort*)

A Ordenação por Pivoteamento *QuickSort* é um algoritmo eficiente de ordenação por divisão e conquista.

Ordenação por Pivoteamento (*QuickSort*)

A Ordenação por Pivoteamento *QuickSort* é um algoritmo eficiente de ordenação por divisão e conquista.

Apesar de ser da mesma classe de complexidade do *MergeSort* e do *HeapSort*, o *QuickSort* é na prática o mais veloz, pois suas constantes são menores. Porém, no pior caso, o *QuickSort* é $O(N^2)$, enquanto que o *MergeSort* e o *HeapSort* são $O(nlog_n)$ para todos os casos. [Cormen et al. 2009]

Ordenação por Pivoteamento (*QuickSort*)

A Ordenação por Pivoteamento *QuickSort* é um algoritmo eficiente de ordenação por divisão e conquista.

Apesar de ser da mesma classe de complexidade do *MergeSort* e do *HeapSort*, o *QuickSort* é na prática o mais veloz, pois suas constantes são menores. Porém, no pior caso, o *QuickSort* é $O(N^2)$, enquanto que o *MergeSort* e o *HeapSort* são $O(nlog_n)$ para todos os casos. [Cormen et al. 2009]

A boa notícia é que há estratégias simples com as quais podemos minimizar as chances de ocorrência do pior caso.

Ordenação por Pivoteamento (*QuickSort*)

O funcionamento do *QuickSort* baseia-se em uma rotina cujo nome é **particionamento**.

Ordenação por Pivoteamento (*QuickSort*)

O funcionamento do *QuickSort* baseia-se em uma rotina cujo nome é **particionamento**.

Particionar significa escolher um número qualquer do vetor, nomeá-lo **pivô**, e colocá-lo em uma posição do vetor tal que todos os elementos à esquerda são menores ou iguais a ele e todos os elementos à direita são maiores que ele.

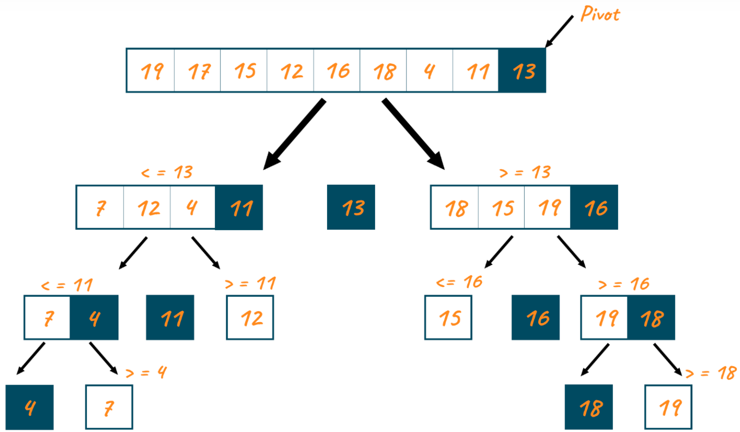
Ordenação por Pivoteamento (*QuickSort*)

O funcionamento do *QuickSort* baseia-se em uma rotina cujo nome é **particionamento**.

Particionar significa escolher um número qualquer do vetor, nomeá-lo **pivô**, e colocá-lo em uma posição do vetor tal que todos os elementos à esquerda são menores ou iguais a ele e todos os elementos à direita são maiores que ele.

Vejamos um vídeo explicativo do algoritmo *QuickSort* no Youtube: `https://www.youtube.com/watch?v=ywWBy6J5gz8`

QuickSort


Pivot
19 17 15 12 16 18 4 11 13
< = 13
7 12 4 11
13
> = 13
18 15 19 16
< = 11
7 4
11
> = 11
12
<= 16
15
16
> = 16
19 18
4
> = 4
7
18
> = 18
19

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

- **Sempre o primeiro número**

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

- **Sempre o primeiro número** $\rightarrow$ E quando estiver ordenado?

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

- **Sempre o primeiro número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o último número**

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

- **Sempre o primeiro número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o último número** $\rightarrow$ E quando estiver ordenado?

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

- **Sempre o primeiro número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o último número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o número do meio do vetor**

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

- **Sempre o primeiro número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o último número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o número do meio do vetor**
- **Sempre um elemento aleatório**

Ordenação por Pivoteamento (*QuickSort*)

O problema do algoritmo *QuickSort* é a escolha do pivô.
Como devemos escolher o pivô?
Estratégias possíveis é escolher:

- **Sempre o primeiro número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o último número** $\rightarrow$ E quando estiver ordenado?
- **Sempre o número do meio do vetor**
- **Sempre um elemento aleatório**

Foi provado que o melhor caso do *QuickSort* acontece quando o seu **pivô** é selecionado sempre de modo **aleatório**. Proporcionando complexidade de tempo de $O(nlog_n)$.

Solução
QuickSort

Ordenação por Pivoteamento → *QuickSort*

Algorithm void quick($int\ vet[], int\ esq, int\ dir$)

$pivo = esq;$
for $i = (esq + 1)$ **to** dir **do**
    $j = i;$
    if $(vet[j] < vet[pivo])$ **then**
        $ch = vet[j];$
        while $(j > pivo)$ **do**
            $vet[j] = vet[j-1];\ j--;$
        end while
        $vet[j] = ch;\ pivo++;$
    end if
end for
CONTINUA $\cdots$

Ordenação por Pivoteamento → *QuickSort*

Algorithm void quick($int\ vet[], int\ esq, int\ dir$)

$\cdots$ CONTINUAÇÃO
if $(pivo - 1 \geq esq)$ **then**
  $quick(vet, esq, pivo - 1);$
end if
if $(pivo + 1 \leq dir)$ **then**
  $quick(vet, pivo + 1, dir);$
end if

Ordenação por Pivoteamento → *QuickSort*

```c
void quick(int vet[], int esq, int dir){
    int pivo = esq, i,ch,j;
    for(i=esq+1;i<=dir;i++){
        j = i;
        if(vet[j] < vet[pivo]){
            ch = vet[j];
            while(j > pivo){
                vet[j] = vet[j-1];
                j--;
            }
            vet[j] = ch;
            pivo++;
        }
    }
    if(pivo-1 >= esq){
        quick(vet,esq,pivo-1);
    }
    if(pivo+1 <= dir){
        quick(vet,pivo+1,dir);
    }
}
```

Solução Modularizada

Ordenação por Pivoteamento → *QuickSort*

Algorithm void troca($int\ vet[], int\ i, int\ j$)

1: $int\ aux = vet[i]$;
2: $vet[i] = vet[j]$;
3: $vet[j] = aux$;

Ordenação por Pivoteamento → *QuickSort*

Algorithm int particiona(*int vet*[], *int inicio*, *int fim*)

```
//O pivô selecionado é sempre o último elemento
pivo = vet[fim];
pivoIndice = inicio;
for i = inicio to fim do
    if (vet[i] ≤ pivo) then
        troca(vet, i, pivoIndice);
        pivoIndice + +;
    end if
end for
troca(vet, pivoIndice, fim);
return pivoIndice;
```

Ordenação por Pivoteamento → *QuickSort*

Algorithm int particionaRandom($int\ vet[], int\ inicio, int\ fim$)

// seleciona um número entre fim e início aleatoriamente
$pivoIndice = (rand()\%(fim - inicio + 1)) + inicio;$
// faz a troca para colocar o pivô no fim
$troca(vet, pivoIndice, fim);$
// chama a função particiona normalmente
$return\ particiona(vet, inicio, fim);$

Ordenação por Pivoteamento → *QuickSort*

Algorithm void quickSort($int\ vet[], int\ inicio, int\ fim$)

if $(inicio < fim)$ **then**
    $pivoIndice = particionaRandom(vet, inicio, fim);$
    $quickSort(vet, inicio, pivoIndice - 1);$
    $quickSort(vet, pivoIndice + 1, fim);$
end if

HeapSort

Ordenação através de Árvores Binárias (*HeapSort*)

Retornaremos para esse algoritmo quando aprendermos o conceito de árvores

Ordenação através de Árvores Binárias (*HeapSort*)

Vejamos um vídeo explicativo do algoritmo *HeapSort* no Youtube:
`https://www.youtube.com/watch?v=Xw2D9aJRBY4`

Ordenação através de Árvores Binárias (*HeapSort*)

O algoritmo, conhecido como *HeapSort*, foi descoberto por *J.W.J. Williams* em 1964.

Ordenação através de Árvores Binárias (*HeapSort*)

O algoritmo, conhecido como *HeapSort*, foi descoberto por *J.W.J. Williams* em 1964.

O *HeapSort* é tem custo linear ($O(nlgn)$), mesmo no pior caso. Suporemos que os índices do vetor são $1 \ldots n$ e não $0 \ldots (n-1)$ (como é usual em C) pois essa convenção torna o código um pouco mais simples.

Ordenação através de Árvores Binárias (*HeapSort*)

O algoritmo, conhecido como *HeapSort*, foi descoberto por *J.W.J. Williams* em 1964.

O *HeapSort* é tem custo linear ($O(nlgn)$), mesmo no pior caso. Suporemos que os índices do vetor são $1 \dots n$ e não $0 \dots (n-1)$ (como é usual em C) pois essa convenção torna o código um pouco mais simples.

Antes de começar a discutir o *HeapSort*, precisamos aprender a enxergar a árvore binária que está escondida em qualquer vetor.

Ordenação através de Árvores Binárias (*HeapSort*)

O conjunto de índices de qualquer vetor $v[1 \ldots m]$ pode ser encarado como uma árvore binária da seguinte maneira:

Ordenação através de Árvores Binárias (*HeapSort*)

O conjunto de índices de qualquer vetor $v[1 \dots m]$ pode ser encarado como uma árvore binária da seguinte maneira:

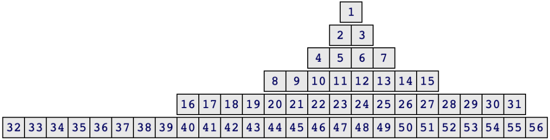
- índice 1 é a raiz da árvore;
- pai de qualquer índice f é $f/2$ (é claro que 1 não tem pai);
- filho esquerdo de um índice p é $2p$ (esse filho só existe se $2p \leq m$);
- filho direito de p é $2p + 1$ (esse filho só existe se $2p + 1 \leq m$).

Ordenação através de Árvores Binárias (*HeapSort*)

Para tornar a árvore binária mais evidente, podemos desenhar o vetor em camadas, de tal modo que cada filho fique na camada seguinte a do pai.

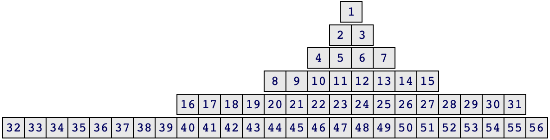
Ordenação através de Árvores Binárias (*HeapSort*)

Para tornar a árvore binária mais evidente, podemos desenhar o vetor em camadas, de tal modo que cada filho fique na camada seguinte a do pai.



Ordenação através de Árvores Binárias (*HeapSort*)

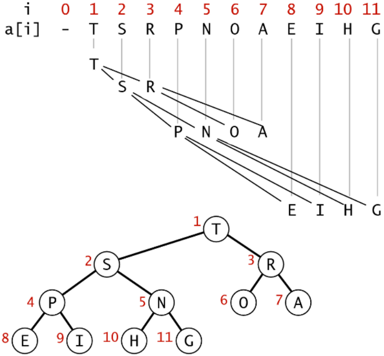
Para tornar a árvore binária mais evidente, podemos desenhar o vetor em camadas, de tal modo que cada filho fique na camada seguinte a do pai.



A figura acima é o desenho do vetor $v[1 \ldots 56]$; os nº nas caixas são os índices i e não os valores $v[i]$. Observe que cada camada, exceto a última, tem duas vezes mais elementos que a anterior. Com isso, o nº de camadas de um vetor $v[1 \ldots m]$ é exatamente $1 + lg(m)$, sendo $lg(m)$ o piso de $log(m)$.

Ordenação através de Árvores Binárias (*HeapSort*)

Para tornar a árvore binária mais evidente, podemos visualisar da seguinte forma:



Heap representations

Ordenação através de Árvores Binárias $\rightarrow$ *HeapSort*

Vejamos uma simulação da árvore do *HeapSort*: $\rightarrow$ `https://www.cs.usfca.edu/~galles/visualization/HeapSort.html`

Ordenação através de Árvores Binárias → *HeapSort*

```
void peneira(int *vet, int raiz, int fundo) {
    int pronto, filhoMax, tmp;

    pronto = 0;
    while ((raiz*2 <= fundo) && (!pronto)) {
        if (raiz*2 == fundo) {
            filhoMax = raiz * 2;
        }
        else if (vet[raiz * 2] > vet[raiz * 2 + 1]) {
            filhoMax = raiz * 2;
        }
        else {
            filhoMax = raiz * 2 + 1;
        }

    if (vet[raiz] < vet[filhoMax]) {
        tmp = vet[raiz];
        vet[raiz] = vet[filhoMax];
        vet[filhoMax] = tmp;
        raiz = filhoMax;
    }
    else {
        pronto = 1;
    }
  }
}
```

Ordenação através de Árvores Binárias → *HeapSort*

```c
void heapsort(int *vet, int n) {
    int i, tmp;

    for (i = (n / 2); i >= 0; i--) {
        peneira(vet, i, n - 1);
    }

    for (i = n-1; i >= 1; i--) {
        tmp = vet[0];
        vet[0] = vet[i];
        vet[i] = tmp;
        peneira(vet, 0, i-1);
    }
}
```

Ordenação através de Árvores Binárias → *HeapSort*

```
int main() {
    int vetor[max], i;

    for(i = 0; i < max; i++){
        printf("Digite o %d° elemento: ",i+1);
        scanf("%d",&vetor[i]);
    }

    heapsort(vetor,max);
    for (i = 0; i < max; i++) {
        printf("%d ", vetor[i]);
    }
    return(0);
}
```

Referências

Referências

CORMEN, T. H. et al. *Introduction to Algorithms*. 2nd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262032937.